

1/11

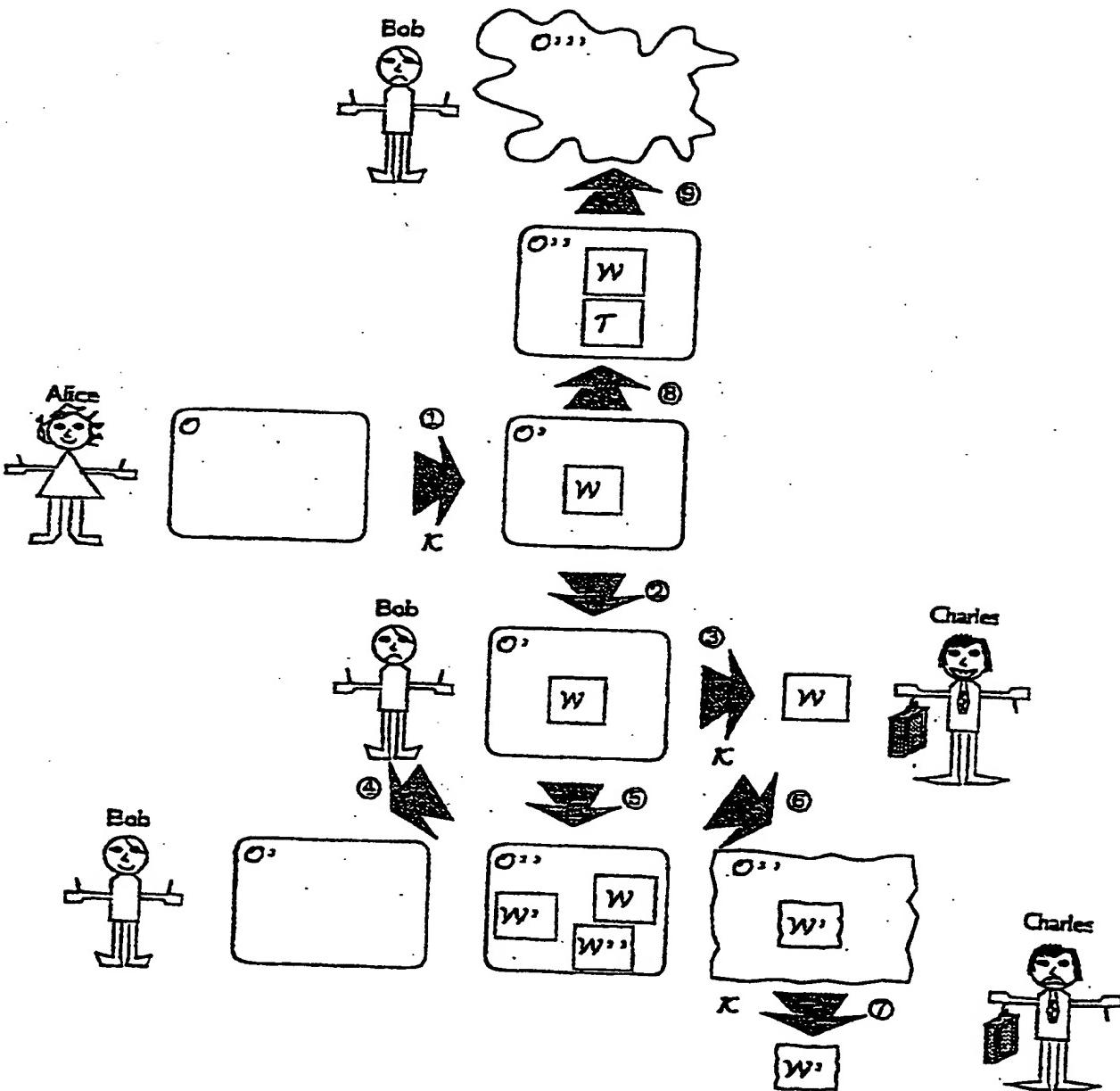
FIGURE 1

Figure 1: At ① Alice adds a watermark W using key K to her object O to make O' . At ② Bob steals a copy of O' . At ③ Charles extracts the watermark from O' using the key K to show that O' is owned by Alice. At ④ Bob successfully removes W from O . At ⑤ Bob adds new watermarks W' and W'' to make it hard for Charles to prove that W is Alice's original watermark. At ⑥ Bob distorts O' (and W) making it difficult for Charles to detect W . At ⑦ Charles attempts to extract the watermark from the distorted object, and either fails completely or gets a distorted watermark. At ⑧ Alice adds tamperproofing T to O . At ⑨ Bob tries to remove W from O , but, due to the tamperproofing, O will be rendered useless to Bob.

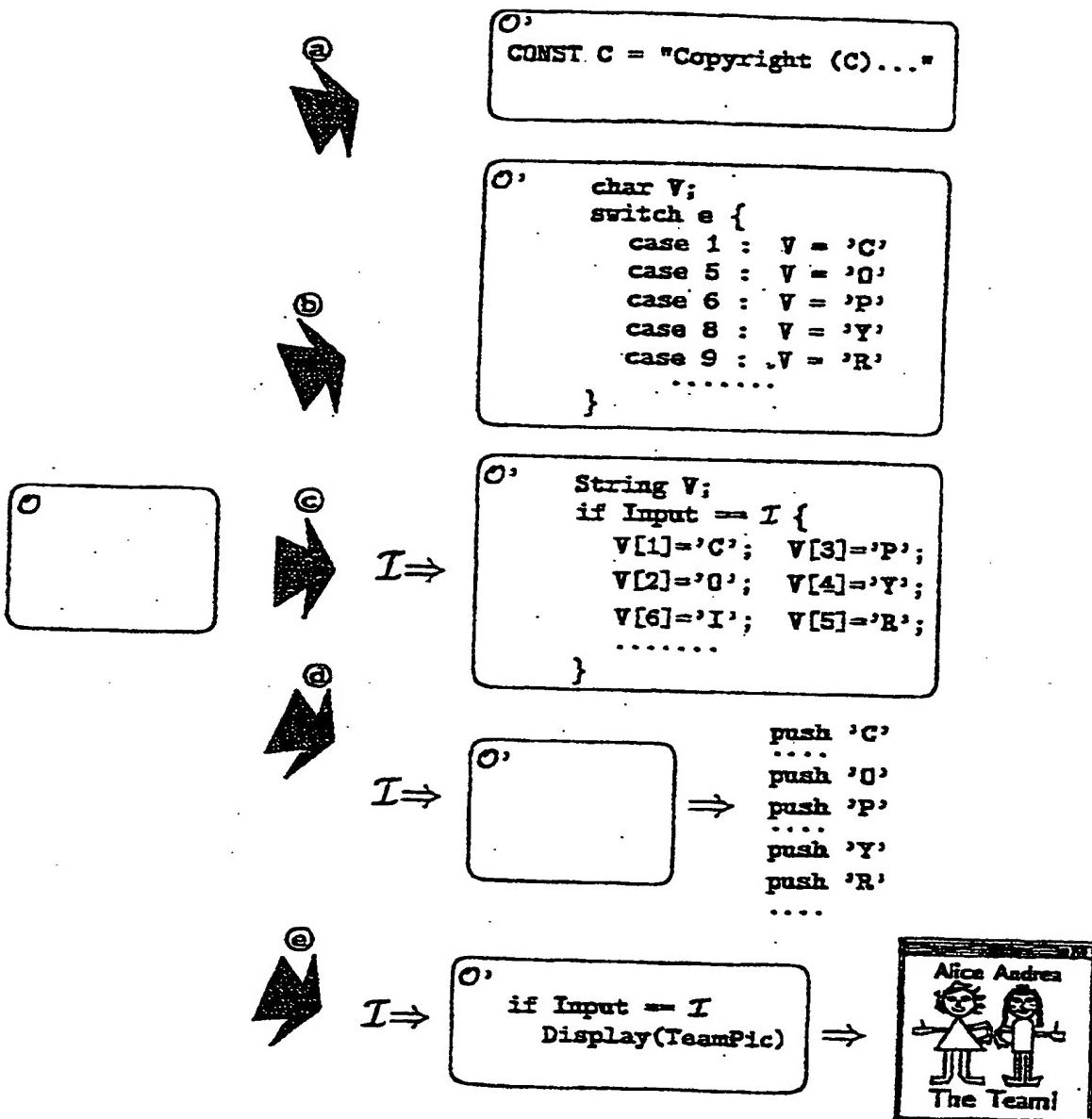


FIGURE 2

Figure 2: In (a) Alice embeds a watermark in the initialized data (string) section of her program. In (b) the watermark is embedded in the text (code) section of the program. In (c) the watermark gets embedded in a global variable V when the program is run with input I . In (d) the watermark is embedded in the execution trace when the program is run with input I . In (e) the watermark is embedded in the unexpected behavior (an “Easter Egg”) of the program when it is run with input I .

3/11

```

String G (int n) {
    int i=0,k;
    String S;
    while (1) {
        L1: if (n==1) {S[i++]= "A"; k=0; goto L6};
        L2: if (n==2) {S[i++]= "B"; k=-2; goto L6};
        L3: if (n==3) {S[i++]= "C"; goto L9};
        L4: if (n==4) {S[i++]= "X"; goto L9};
        L5: if (n==5) {S[i++]= "C"; goto L11};
            if (n>12) goto L1;
        L6: if (k++<=2) {S[i++]= "A"; goto L6} else goto L8;
        L8: return S;
        L9: S[i++]= "C"; goto L10;
        L10: S[i++]= "B"; goto L8;
        L11: S[i++]= "C"; goto L12;
        L12: goto L10;
    }
}

```

FIGURE 3

Figure 3: A function producing the strings "AAA", "AAAAA", "XCB", and "CCB".

DRAFT - 566674260

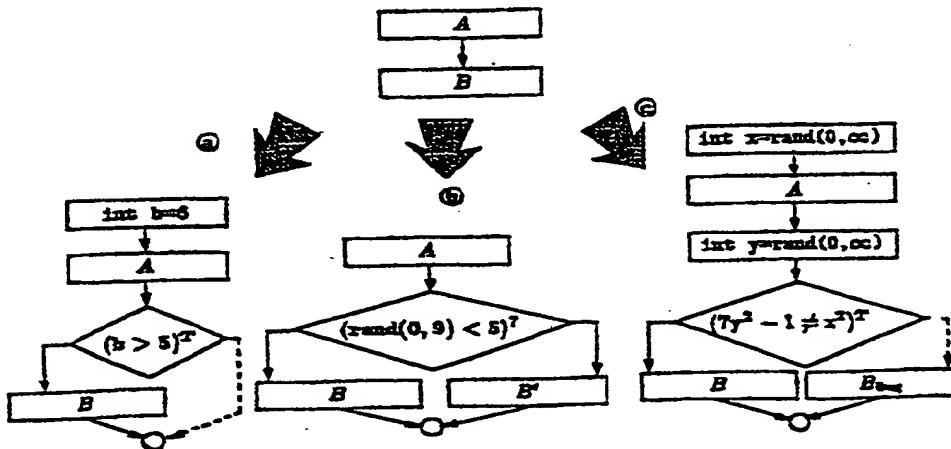
FIGURE 4

Figure 4: Inserting bogus predicates in a program. In ① an opaque predicate $b > 5^T$ is inserted. This predicate is always true. In ② an opaque predicate $\text{rand}(0, 9) < 5^T$ is inserted. This predicate is sometimes true (in which case B is executed), and sometimes false (in which case an obfuscated version of B is executed). In ③ an opaque true predicate is inserted. This predicate appears to sometimes execute an obfuscated buggy version of B , but, in fact, never does.

4/11

$g(V)$		$f(p, q)$	V	$2p + q$	$\text{AND}[A, B]$		A			
p	q				0	1	2	3		
0	0	False		0		0	3	0	0	0
0	1	True		1	B	1	3	1	2	3
1	0	True		2		2	0	2	1	3
1	1	False		3		3	3	0	0	3

- (1) bool A,B,C;
(2) B = False;
(3) C = False;
(4) C = A & B;
(5) C = A & B;
(6) if (A) ...;
(7) if (B) ...;
- (1') short a1,a2,b1,b2,c1,c2;
(2') b1=0; b2=0;
(3') c1=1; c2=1;
(4') x=AND[2*a1+a2, 2*b1+b2]; c1=x/2; c2=x%2;
(5') c1=(a1 ^ a2) & (b1 ^ b2); c2=0;
(6') x=2*a1+a2; if ((x==1) || (x==2)) ...;
(7') if (b1 ^ b2) ...;

FIGURE 5

Figure 5: Variable splitting example. We show one possible choice of representation for split boolean variables. The table indicates that boolean variable V has been split into two short integer variables p and q . If $p = q = 0$ or $p = q = 1$ then V is False, otherwise, V is True. Given this new representation, we devise substitutions for the built-in boolean operations. In the example, we provide a run-time lookup table for each operator. Given two boolean variables $V_1 = [p, q]$ and $V_2 = [r, s]$, " $V_1 \& V_2$ " is computed as "AND[$2p + q, 2r + s$]".

$$\begin{aligned}
Z(X+r, Y) &= 2^{32} \cdot Y + (r+X) &= Z(X, Y) + r \\
Z(X, Y+r) &= 2^{32} \cdot (Y+r) + X &= Z(X, Y) + r \cdot 2^{32} \\
Z(X \cdot r, Y) &= 2^{32} \cdot Y + X \cdot r &= Z(X, Y) + (r-1) \cdot X \\
Z(X, Y \cdot r) &= 2^{32} \cdot Y \cdot r + X &= Z(X, Y) + (r-1) \cdot 2^{32} \cdot Y
\end{aligned}$$

- (1) int X=45;
int Y=95;
(2) X += 5;
(3) Y += 11; \Rightarrow (2') Z += 5;
(4) X += c;
(5) Y += d;
- (1') long Z=167759086119551045;
(3') Z += 47244640256;
(4') Z += (c-1)*(Z & 4294967295);
(5') Z += (d-1)*(Z & 18446744069414584320);

FIGURE 6

Figure 6: Merging two 32-bit variables X and Y into one 64-bit variable Z . Y occupies the top 32 bits of Z , X the bottom 32 bits. If the actual range of either X or Y can be deduced from the program, less intuitive merges could be used. First we give rules for addition and multiplication with X and Y , then show some simple examples.

5/11

```

int Sum(int A[]) {
    int sum=0, i=0, pc=0;
    int s[] = new int [5], sp=-1;
    loop: while (true)
        switch("fcgabcd".charAt(pc)) {
            case 'a': sum += s[sp--]; pc++; break;
            case 'b': i++; pc++; break;
            case 'c': s[++sp] = i; pc++; break;
            case 'd': if (s[sp-1] > s[sp]) pc -= 6;
                        else break loop; break;
            case 'e': s[++sp] = A.length; pc++; break;
            case 'f': pc += 5; break;
            case 'g': s[sp] = A[s[sp]]; pc++; break;
        }
    return sum;
}

```

FIGURE 7

Figure 7: The Java method `Sum` on the left is obfuscated by translating it into the bytecode "fcgabcd". This code is then executed by a stack-based interpreter specialized to handle this particular virtual machine code. This technique is similar to Probsting's superoperators [20].

6/11

TOSSED = 69807250

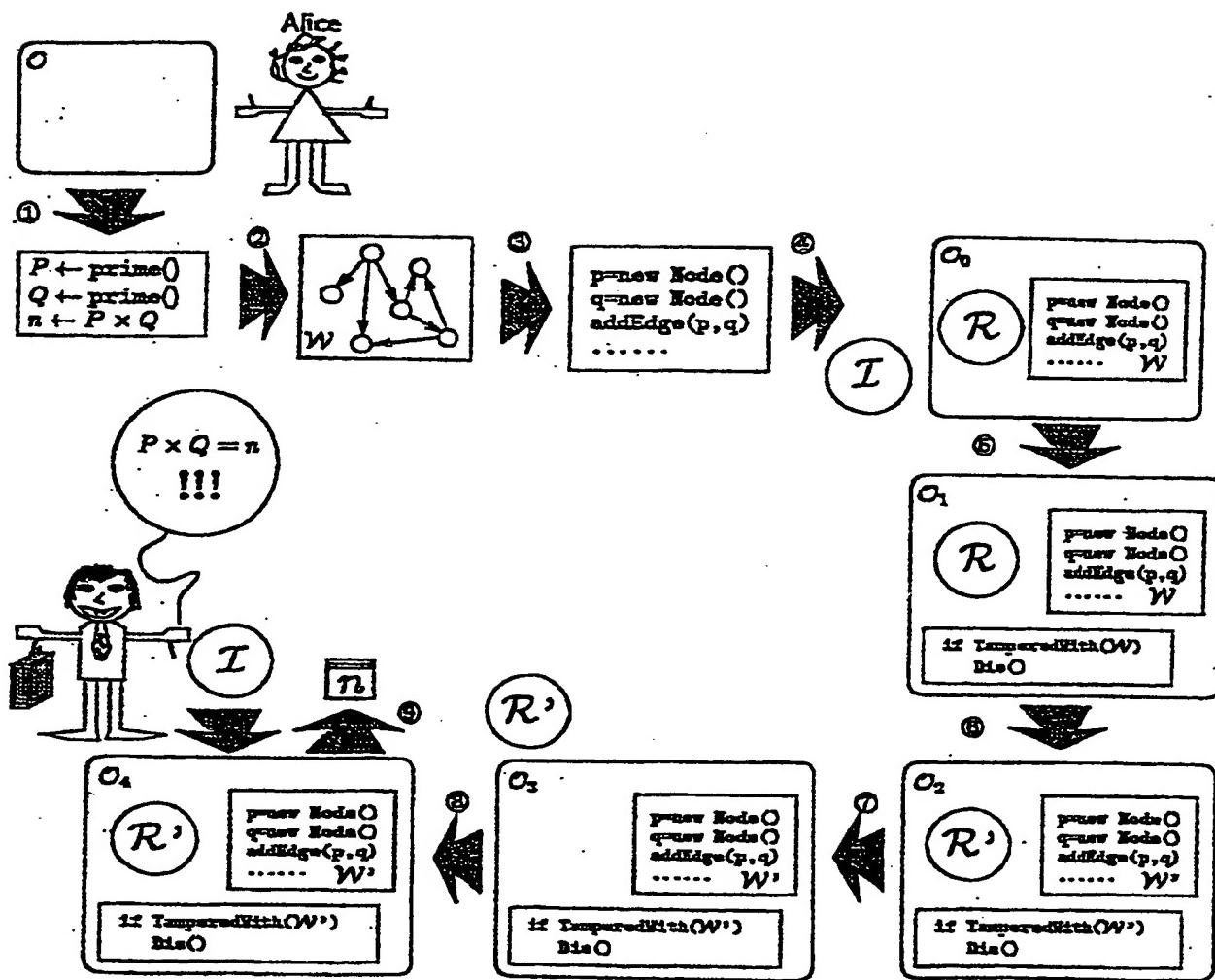
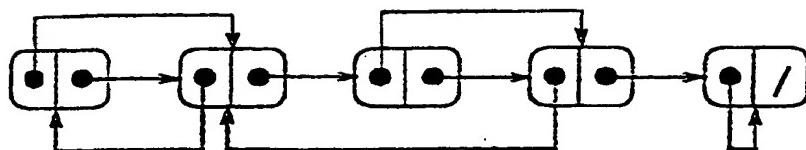
FIGURE 8

Figure 8: At ① Alice selects two large primes P and Q , and computes their product n . At ② she embeds n in the topology of a graph. This graph is her watermark W . At ③ W is converted to a program which builds the graph. At ④ the program is embedded into the original program O_0 , such that when O_0 is run with I as input, W is built. Also, a recognizer program R is constructed, which is able to identify W on the heap, and extract n from it. At ⑤ tamperproofing is added, to prevent the graph from being obfuscated to such an extent that R cannot identify it. At ⑥ the application (including the watermark, tamperproofing code, and recognizer) is obfuscated. At ⑦ the recognizer is removed from the application. O_3 is the version of Alice's program that is distributed. At ⑧ Charles links in the recognizer program R with O_3 . At ⑨ the application is run with I as input, and the recognizer R produces n . Since Charles is the only one who can factor n , he can prove the legal origin of Alice's program.

7/11



$$3 \cdot 6^4 + 2 \cdot 6^3 + 3 \cdot 6^2 + 4 \cdot 6^1 + 1 \cdot 6^0 = 4453 = 61 \cdot 73$$

FIGURE 9

Figure 9: Embedding a watermark into a graph structure. The structure is essentially a linked list. The rightmost pointer of each node is the next field, while the second field encodes a digit. In this example, 0=null (/), 1=a self-pointer, 2=a one-step back pointer, 3=a one step forward pointer, 4=a two step back pointer, and 5=a 2 step forward pointer. This allows us to encode a value $61 * 73 = 4453_{10}$ as the base-6 value 32341_6 .

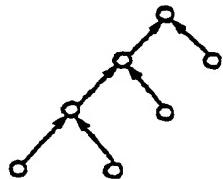
FIGURE 10

Figure 10: The twenty-second tree in an enumeration of the oriented trees with seven vertices.

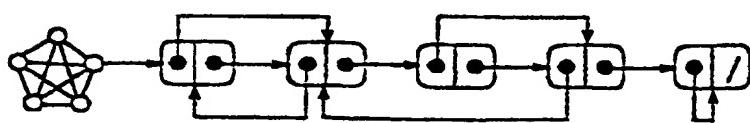
FIGURE 11

Figure 11: A 5-clique is used to mark the beginning of an encoded value.
SUBSTITUE SHEET (Rule 26)

8/11

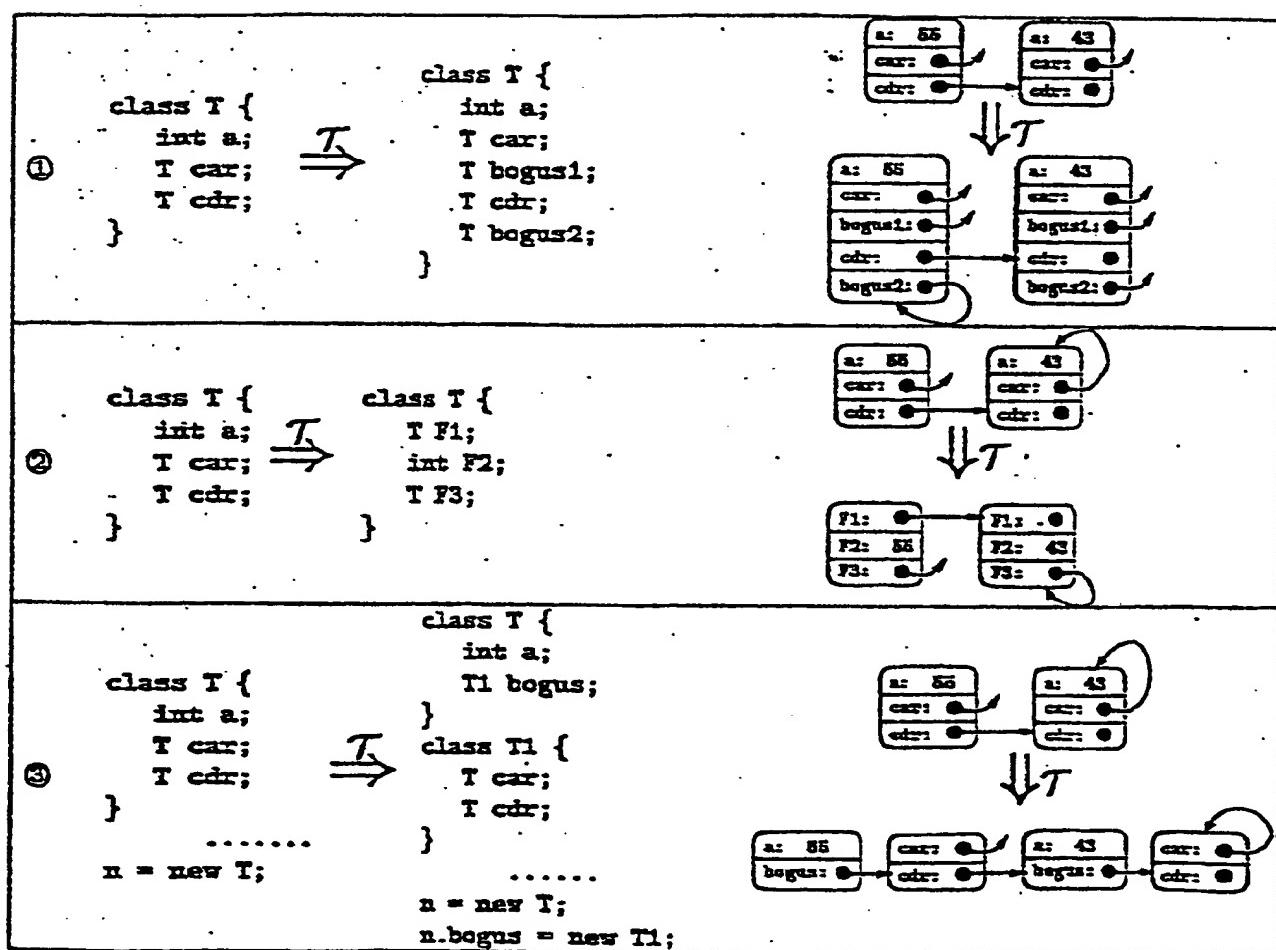
FIGURE 12

Figure 12: Obfuscation of dynamic structures. In ① we add bogus pointer fields to all nodes of type T. In ② we rename and reorder fields. In ③ we add a level of indirection by splitting all nodes in two.

9/11

```

class C {public int a; public C car, cir;}

public static void main(String[] args) {
    Field[] F = C.class.getFields();
    if (F.length != 3)
        die();
    if (F[0].getType() != java.lang.Integer.TYPE)
        die();
    if (F[1].getType() != C.class)
        die();
    if (F[2].getType() != C.class)
        die();
}

```

(a)

```

class C {public int a; public C car, cir;}

public static void main(String[] args)
    throws NoSuchFieldException,
    IllegalAccessException {
    Field f;
    String V;
    C n = new C();
    Class c = n.getClass();
    if (PF) {
        f = c.getField(V=><car>);
        ① f.set(n, null);
    }
    Field F = c.getFields();
    int R;
    ② F[R=1].set(n, n.car);
}

```

(b)

FIGURE 13

Figure 13: Examples of tamperproofing Java code using the reflection interface.

09/719399

WO 99/64973

PCT/NZ99/00081

10/11

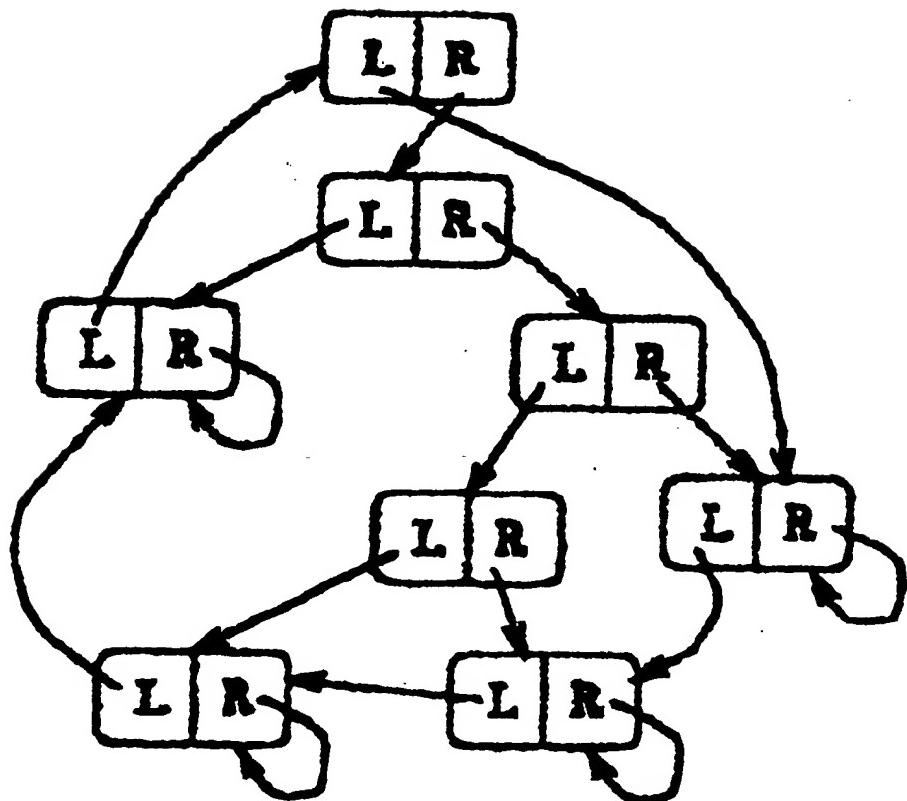


FIG 14

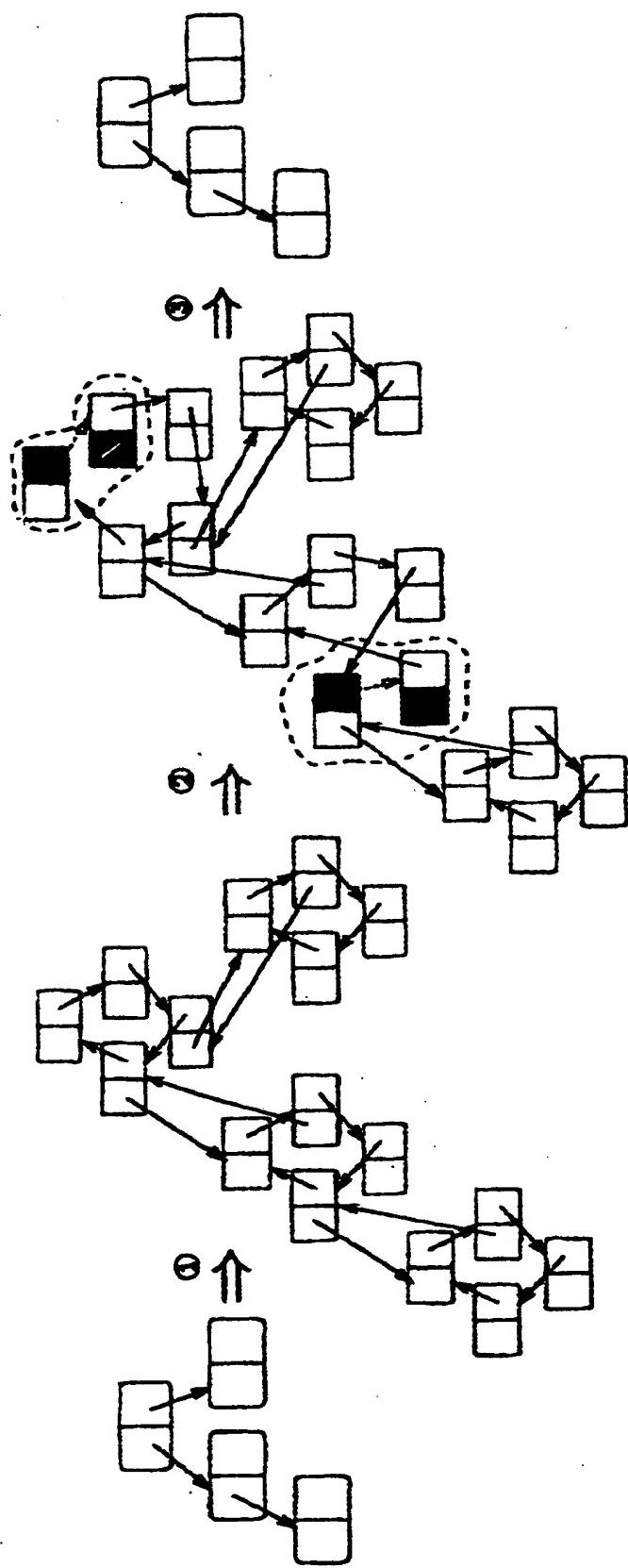


Figure 15 Tamperproofing against node-splitting. At ① we expand each node of our original watermark tree into a 4-cycle. At ② an adversary splits two nodes. The structure of the graph ensures that these nodes will fall on a cycle. At ③ the recognizer shrinks the biconnected components of the underlying (undirected) graph. The result is a graph isomorphic to our original watermark.

FIG 15